САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

КАФЕДРА ТЕХНОЛОГИИ ПРОГРАММИРОВАНИЯ

Лехова Ирина Андреевна

Выпускная квалификационная работа бакалавра

“Поиск дубликатов изображений на примере Instagram”

Направление 010300

Фундаментальная информатика и информационные технологии

Научный руководитель,

старший преподаватель

Уланов А.В.

Санкт-Петербург

2016

Содержание

Введение

В нашем обществе использование чужих идей с указанием источника или официально приобретенных продуктов не запрещается. Но существует проблема нарушения авторских прав, незаконного использования, обусловленная прогрессом современных информационных технологий и широким использованием сети Интернет. Интернет помогает стремительно развиваться не только авторству, но и плагиату. В настоящее время в большинстве стран мира существуют законы, защищающие авторские права. Несоблюдение этих законов может приводить к серьёзным последствиям, вплоть до тюремного заключения.

Плагиат фотографий является не меньшей проблемой чем, например, плагиат авторских текстов. Данная проблема весьма насущна для Instragram - социальной сети для обмена фотографиями и видеозаписями, которая достигла огромной популярности за последние 5 лет. Кража и повторная публикация чужих фотографий в Instagram является не только нарушением авторских прав, но и кражей личной собственности.

Целью данной выпускной квалификационной работы является создание прототипа системы поиска дубликатов изображений на примере социальной сети Instagram. Подобная система будет полезна людям, которые беспокоятся о том, что кто-то присваивает их авторство себе, и людям, которые хотят получить информацию о первоисточнике фотографии. Также она окажет большое содействие в борьбе со спамом – навязчивой нежелательной рекламой.

Постановка задачи

Для достижения цели, указанной в введении ВКР, были поставлены задачи:

1. Ознакомиться с уже существующими системами поиска дубликатов изображений и провести их обзор.
2. Рассмотреть возможные способы решения задачи и описать алгоритмы, наиболее подходящие для ее решения.
3. Реализовать рассмотренные алгоритмы, провести сравнительные тесты.
4. Выбрать наиболее эффективный алгоритм для дальнейшей работы
5. Создать веб-сервис по поиску похожих изображений на базе Microsoft Azure

Используемые термины

Instagram - социальная сеть для обмена фотографиями и видеозаписями

Спам - массовая рассылка коммерческой и иной рекламы или подобных коммерческих видов сообщений лицам, нежелающим их получать.

Веб-сервис - услуги, предоставляемые в интернете

Хэш-функция - алгоритм, преобразующий строку произвольной длины в битовую строку фиксированной длины.

Хэш изображения - битовая строка фиксированной длины, полученная в результате преобразования изображения по определенному алгоритму.

Расстояние Хэмминга - число позиций, в которых соответствующие цифры двух двоичных слов одинаковой длины различны. [6]

Пиксель - наименьший элемент 2D-изображения в растровой графике

Градации серого - отображение изображения в оттенках серого цвета.

Цветовой баланс - соотношение цветов в изображении.

Гамма-коррекция - коррекция яркости цифрового изображения.

Гистограмма изображения - график распределения элементов изображения с различной яркостью, в котором по горизонтали располагается яркость, а по вертикали - число пикселей с заданным значением яркости.

Шум - дефект изображения, ухудшающий его качество.

Размытие - смешивание двух цветов из палитры для имитирования третьего.

Яркость - количество белого цвета на изображении.

Контрастность - разница между разными, расположенными близко цветами.

Масштабирование - изменение размера изображения.

Коллизия - ложные срабатывания алгоритма. В данной работе - ситуации, когда у двух разных изображений два идентичных или похожих хэшей.

Облачная платформа - модель, когда потребителю предоставляется возможность использования облачных технологий для размещения базового программного обеспечения для последующего размещения на нём своих приложений.

База данных – совокупность материалов, хранящихся таким способом, чтобы они могли быть найдены и обработаны.

Обзор литературы

При написании данной работы были использованы статьи из материалов международных конференций (Intelligent Robotics and Applications: 8th International Conference, IEEE International Conference on Image Processing), статьи из интернета и магистерская диссертация Christoph Zauner.

В процессе поиска литературы, которая могла бы быть полезной для данной работы, автор столкнулся с проблемой отсутствия достойного материала об алгоритмах перцептуального хэширования на русском языке. Это обусловлено тем, что данная тема не пользуется популярностью среди российских ученых, нежели в других странах. Тем не менее, в рунете доступен поверхностный материал в виде статей на ресурсе, посвященном IT-технологиям, Habrahabr[1], но данная информация полезна лишь в целях ознакомления с базовыми понятиями хэширования изображений, так как она не подкреплена никакими практическими результатами или ссылками на научные труды.

Статьи Neal Krawetz ”Kind of Like That” [1] и ”Looks Like IT” [2], доступные в блоге The Hacker Factor Blog[2] часто цитируются в научных трудах других людей. В этих публикациях достаточно подробно и понятно изложены фундаментальные основы перцептуальных хэшей и принципы работы алгоритмов aHash, pHash DCT и dHash. Несмотря на то, что данный сайт подобен вышеописанному Habrahabr, у этих статей есть большое преимущество – в них читателя впервые знакомят с алгоритмом перцептуального хэширования dHash, разработанным David Oftedal, который посодействовал написанию статьи “Kind of Like That”.

[1] https://habrahabr.ru/
[2] http://www.hackerfactor.com/blog

Еще одним из основных источников литературы является магистерская диссертация Christoph Zauner Implementation and Benchmarking of Perceptual Image Hash Functions [5], которая также часто цитируема. В своей работе автор достаточно подробно рассказывает о том, что такое перцептуальные хэш-функции, как они применимы к задаче сравнения изображений, проводит достаточно подробный обзор возможных модификаций алгоритма pHash и описывает работу по созданию собственного фрэймворка Rihamark, который предоставляет возможность проводить анализ хэш-функций с помощью встроенных инструментов для модификаций изображений и проведения тестов.

Также было подмечено, что в большинстве научных публикаций и статьях в интернете не различают понятия Perceptual(Перцептуальный) и Perceptive(Перцептивный). Это является достаточно грубой ошибкой, так как изначально первое понятие относилось к названию класса изучаемых в этой работе алгоритмов, а второе – к конкретному алгоритму, pHash (Perceptive hash или Перцептивный хэш).

Глава 1. Примеры существующих систем поиска дубликатов изображений

На сегодняшний день существует множество систем поиска похожих изображений, имеющих определенные возможности и основанных на различных алгоритмах. Рассмотрим некоторые из них.

1.1 Google Images

Рисунок 1.1 - Логотип Google Images и его интерфейс

Google Images - специальный сервис одной из крупнейших поисковых систем Google. Он использует обратный поиск изображения и позволяет пользователям найти похожие изображения просто путем загрузки изображения или его прямой ссылки. Google решает эту задачу путем анализа представленного изображения и построения его математической модели с помощью современных алгоритмов. Далее происходит сравнение с миллиардами других изображений в базах данных Google и выбираются похожие. Точность поиска выше, если искомое изображение является популярным, т.е. его уже искали до этого. Кроме того, Google Images будет предлагать "лучшее предположение" для этого изображения на основе описательных метаданных результатов.

1.2 Яндекс.Картинки

Яндекс.Картинки - аналогичный сервис, предоставляемый российской поисковой системой Яндекс. Также специализируется на нахождении не только точных копий изображения, но и просто похожих. Взаимодействия пользователя с сервисом происходит аналогично Google Images.

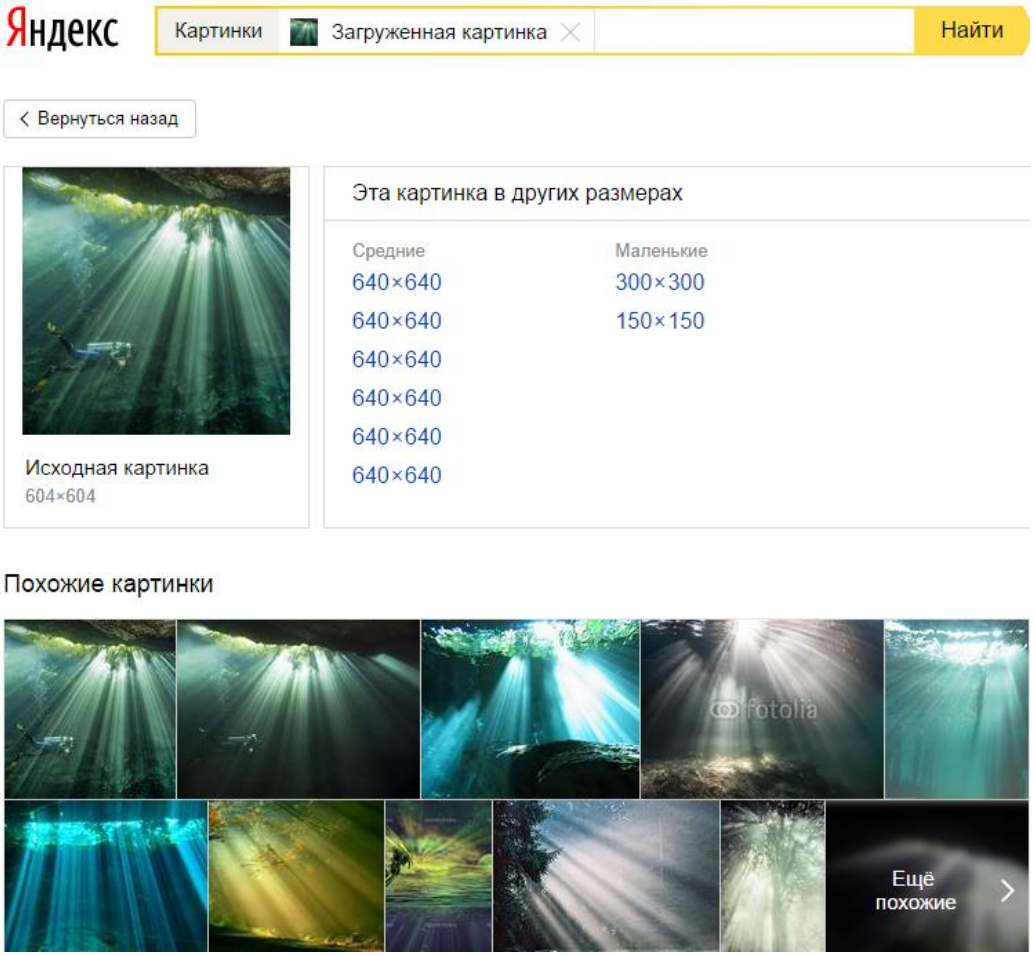

Рисунок 1.2 - Процесс поиска похожих изображений в Яндекс.Картинках

Поиск изображений основан на собственной технологии Яндекса - Сибирь (от англ. CBIR – Content-based image retrieval, рус. Поиск изображения по содержанию). Поиск практически идентичных изображений (та же картинка, но другого размера, с нанесением небольшого копирайта и пр.) происходит следующим образом: алгоритм разбивает загруженную картинку на визуальные слова и с их помощью сопоставляет её с миллиардами известных ему изображений, отбирая дубликаты. Для поиска похожих изображений, но не практически идентичных, “Сибирь” использует глубокие нейронные сети. [9]

1.3 TinEye

TinEye - это поисковая система, специализирующаяся на поиске в интернете изображений, похожих на изображение-образец. То есть, фактически, она находит “почти” дубликаты, но не может осуществлять, например, поиск картинок аналогичного содержания.

Ресурс TinEye открывает для пользователей ряд различных возможностей. В частности, с его помощью можно:

- Узнать автора изображения или фотографии;
- Отследить, когда изображение появилось впервые в Интернете;
- Найти то же изображение/фотографию, но с более высоким разрешением, либо каким-то образом отредактированное.

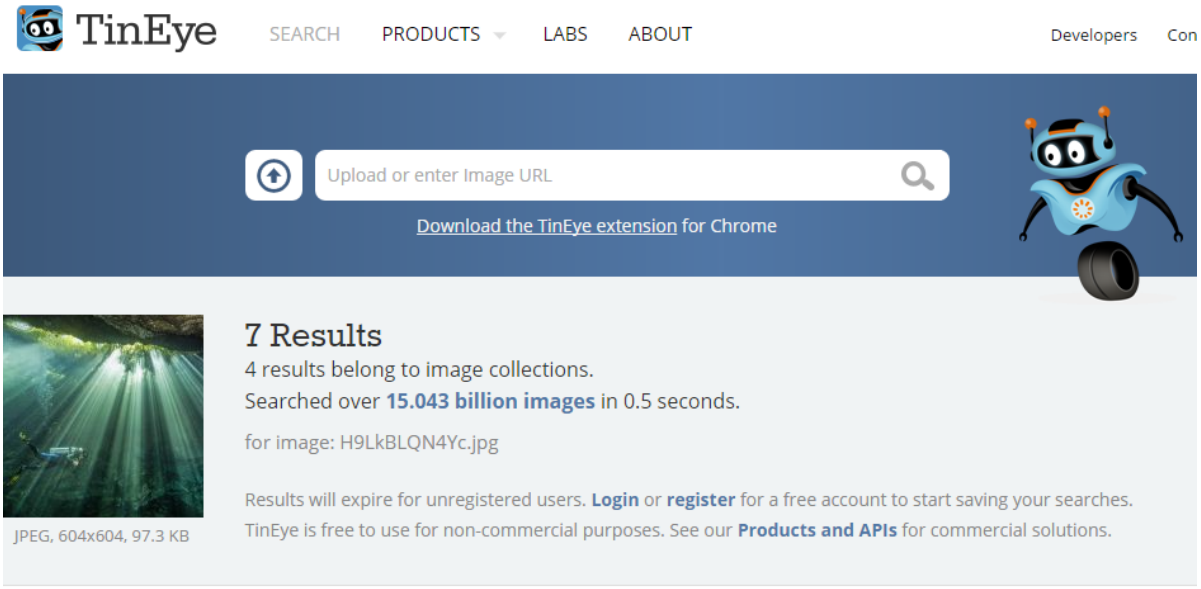

Рисунок 1.3 - Процесс поиска похожих изображений в TinEye

В отличие от большинства поисковых систем, TinEye не использует имена изображений или любые метаданные для выполнения поиска. После того как пользователь загрузил фотографию для поиска, TinEye создает уникальный и компактный цифровой “отпечаток” изображения, а затем сравнивает этот отпечаток с биллионами изображений в индексе TinEye.

К сожалению, создатели данного сервиса не раскрывают технологии, которые были использованы для его реализации, но результаты поиска

плагиата похожи на результаты тех алгоритмов, о которых рассказывается далее в данной работе.

1.4 ReportStatistics

Еще одним примером похожей системы является бот RepostStatistics, работающий на базе Imgur – онлайн-сервиса загрузки, хранения и обмена фотоизображений.

Задача ReportStatistics состоит в том, чтобы самостоятельно находить дубликаты фотографий и оставлять комментарии под новыми изображениями с ссылкой на оригинал. В основе его работы лежат реализованные на языке программирования python алгоритмы aHash и dHash, о которых будет рассказано далее.

Глава 2. Перцептуальные хэш-алгоритмы

Одним из подходов решения задачи распознавания графических образов является использование нейронных сетей. Предварительно сеть обучается с помощью тренировочного множества, далее на вход подаются изображения, которые нужно охарактеризовать как дубликат или не дубликат. Главное преимущество использования нейронных систем в поиске дубликатов в возможности находить не только копии изображений, но и просто похожие (например, фотографии, снятые с другого ракурса или аналогичное содержание). Но, при использовании данного подхода для поиска спама или плагиата, возникает проблема: для каждого типа спам-изображения необходимо большое количество обучающих данных, что трудно в плане реализации, потому что невозможно знать заранее, на какой фотографии может появиться спам или плагиат.

Также сравнение двух изображений можно осуществить с помощью побитового сравнения двух файлов. Это один из самых первых способов сравнения изображений. В настоящее время он достаточно устаревший, существует огромное количество альтернативных способов решения данной проблемы, но, на момент своего создания, он являлся единственным решением задачи. Его главный недостаток заключается в том, что чем больше файлов для сравнения, тем больше потребуется вычислительных затрат.

Более эффективными в решении данной задачи являются алгоритмы перцептуального хэширования. Вкратце, перцептуальный хэш — это свертка каких-то признаков, описывающих картинку. Эти алгоритмы объединяет то, что сперва происходит преобразование изображений (изменение в размере, перевод в градации серого и т.д. – все это зависит от конкретного алгоритма), строятся хэши изображений, а потом хэши двух изображений сравниваются

между собой. Основные различия именно в хэш-функциях. Размер хэша равен количеству пикселей преобразованного изображения.

Для сравнения хэшей алгоритмы, описываемые в данной работе, используют расстояние Хэмминга, которое достаточно просто для вычисления. Чем меньше это расстояние, тем более вероятно, что изображения одинаковы. И наоборот, чем больше дистанция, тем больше отличие.

Таким образом, задача сравнения изображений сводится к вычислению их хэш-значений и нахождению расстояния Хэмминга между хэшами. Алгоритмы этого класса просты как для реализации, так и для поиска похожих изображений по базе данных путем сравнения хэщей. Рассмотрим самые известные из них:

2.1 aHash (Average hash или простой перцептуальный хэш)

aHash - самый легкий для реализации из описываемых в этой главе алгоритмов. Его отличительной чертой является то, что его работа основана на проверке на среднее значение по всем точкам изображения.

Шаги:

1. Уменьшить размер. Чем больше уменьшенное изображение, тем более точным будет результат, но на это потребуется больше времени. Обычно при реализации этого алгоритма изображение сжимается до 8x8 (здесь и далее в пикселях). Данная операция выполняется для избавления от высоких частот.

2. Убрать цвет. Сжатое изображение переводится в градации серого, в результате чего размер хэша становится в 3 раза меньше. (происходит уменьшения количества цветовых компонент с трех (в модели RGB) до одной уровня серого). Преобразуется по формуле:

$$img[i,j] = \frac{R+G+B}{3} \quad ,$$

где $img[i,j]$ – новое значение [i, j] - пикселя изображения,

R, G, B - значения цветов пикселя в пространстве RGB.

3. Найти среднее. Необходимо вычислить среднее значение по всем 64 точкам изображения.

$$avg = \frac{\sum_{i=1}^{8}\sum_{j=1}^{8} img[i,j]}{8 \times 8}$$

4. Построить битовую цепочку. В зависимости от того, является ли значение пикселя больше среднего или меньше, в конец цепочки записывается 1 или 0.

$$b[i * j] = 1, если\ img[i,j] \geq avg$$

$$b[i * j] = 0, если\ img[i,j] < avg$$

5. Построить хэш. Необходимо перевести 64 отдельных бита в одно 64-битное значение. Порядок не имеет значения, но принято записывать биты начиная с левого верхнего угла и заканчивая правым нижним.

Данный алгоритм достаточно быстрый, чувствителен к операциям, изменяющим среднее значение - изменения цветового баланса или уровней, так как он основан на средних значениях цветов.

2.2 pHash (Perceptive Hash или Перцептивный хэш)

pHash во многом повторяет шаги aHash, но кроме того еще выполняется DCT (Discrete Cosine transform) - дискретное косинусное преобразование, которое делит исходное изображение на гармоники дискретного сигнала, влияющие на качество изображения.

Шаги:

1. Требуется уменьшить изображение до 32x32.

2. Провести перевод изображения в оттенки серого

3. Провести DCT. Оно разбивает изображение на набор частот и векторов.

$$F(u,v) = \left(\frac{2}{N}\right)^{\frac{1}{2}}\left(\frac{2}{M}\right)^{\frac{1}{2}}\sum_{i=0}^{N-1}\sum_{j=0}^{M-1} \Lambda(i)\cdot\Lambda(j)\cdot\cos\left[\frac{\pi\cdot u}{2\cdot N}(2i+1)\right]\cdot\cos\left[\frac{\pi\cdot v}{2\cdot M}(2j+1)\right]\cdot f(i,j)\ ,$$

где:

$$\Lambda(\xi) = \frac{1}{\sqrt{2}}\ , если\ \xi = 0$$

$$\Lambda(\xi) = 1, если\ \xi \neq 0$$

- M, N — определяют размер входной матрицы. (32x32)
- f (i, j) — значение интенсивности пикселя (i, j)
- F (u, v) —коэффициент из матрицы DCT, располагающийся на строке u и столбце v. [3]

4. Сократить DCT. Требуется сохранить только левый верхний блок 8x8. Нам необходимы низкочастотные компоненты изображения. Высокочастотные компоненты, располагающиеся ближе к правому нижнему углу, не несут особой ценности.

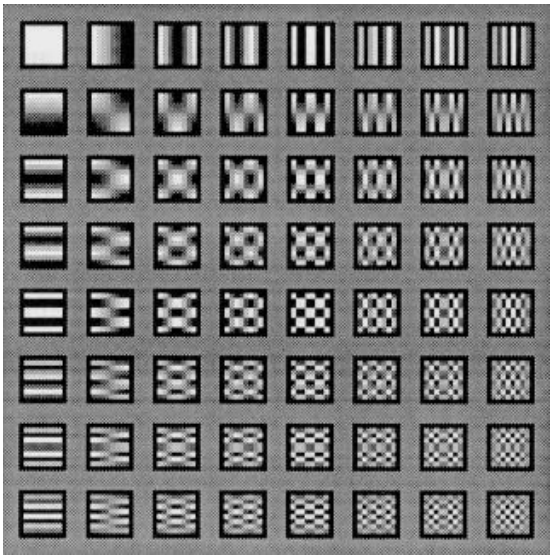

Рисунок 2.1 - Разбиение изображения на частотные компоненты

5. Вычислить среднее значение. На этом шаге требуется убрать из расчёта самый первый коэффициент, чтобы исключить из описания хэша пустую информацию, например, одинаковые цвета. [3]

6. Присвоить каждому из 64 DCT-значений 0 или 1 в зависимости от того, больше оно или меньше среднего значения.

7. Построить хэш. 64 бита превращаются в 64-битное значение.

Такой алгоритм уже способен выдержать изменение гистограммы изображения или его гамма-коррекцию. PHash нечувствителен к изменению контрастности, яркости и к масштабированию.

2.3 dHash (Difference Hash)

Создателем данного алгоритма является David Oftedal. dHash прост в реализации и обладает высокой скоростью и точностью работы. Основан на отслеживании градиента изображения.

Шаги:

1. Уменьшить размер изображения до 9x8 (в общем случае N + 1 x N).
2. Перевести изображение в оттенки серого
3. Вычислить разницу между следующим и предыдущим пикселем. В результате получается матрица 8x8
4. Построить битовую цепочку. Если значение текущего пикселя больше предыдущего, значение хэша принимается 1, в противном случае 0.
5. Построить хэш. Требуется перевести 64 отдельных бита в одно 64-битное значение.

2.4 gHash (Gradient Hash)

Также была проведена работа по созданию собственного алгоритма перцептуального хэширования – gHash, градиентного хэша. За основу был взят алгоритм dHash. Опытным путем было выявлено, что отслеживание градиента изображения по столбцам и строкам не менее эффективно попиксельного метода, лежащем в основе dHash.

Шаги:

1. Уменьшить размер изображения до 32x32.
2. Перевести изображение в оттенки серого
3. Построить битовую цепочку. Каждому столбцу присваивается 1 или 0 в зависимости от того, больше или меньше сумма значений его пикселей в сравнении со следующим столбцом. Далее происходит запись этого значения в конец битовой строки. Аналогичные

операции проводятся со строками. В результате получается цепочка из 64 битов.

4. Построить хэш. Требуется перевести 64 отдельных бита в одно 64-битное значение.

Если допустить, что изображение уменьшается до $n \times n$ пикселей, то, пренебрегая способом масштабирования изображения, сложности алгоритмов aHash, pHash, dHash и gHash, соответственно: $O(n^2)$, $O(n^4)$, $O(n^2)$ и $O(n^2)$. В оценке сложности алгоритма pHash большую роль играют затраты на вычисления DCT.

Глава 3. Реализация алгоритмов

Для реализации алгоритмов gHash, dHash и aHash был выбран объектно-ориентированный язык программирования C# со встроенной библиотекой для обработки и прорисовки изображений System.Drawing и готовая библиотека pHash[3] для алгоритма pHash, так как дискретное косинусное преобразование достаточно сложное для реализации

Для анализа и тестирования работы скорости алгоритмов использовалась выборка из 8000 уникальных фотографий из инстаграма natgeo[4].

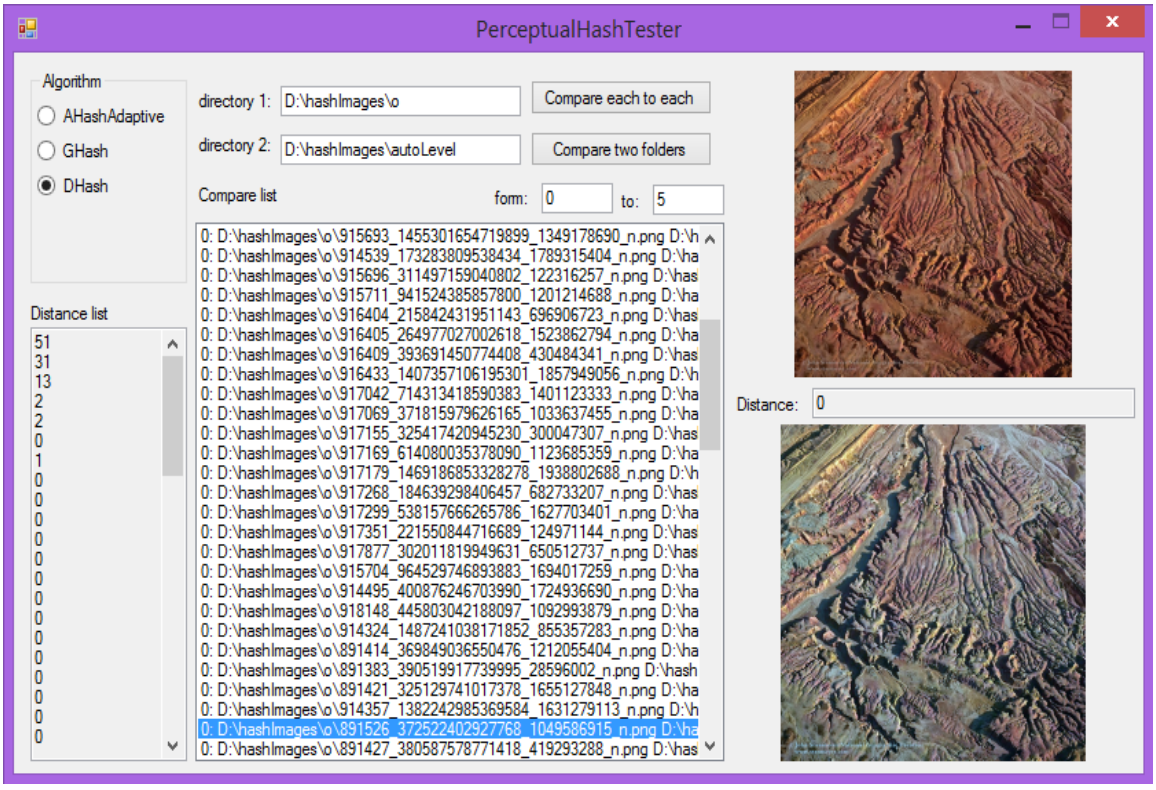

Рисунок 3.1 - Пример работы программы

На первом этапе для каждого алгоритма был проведен замер времени, необходимого для обработки 8000 фотографий. Результаты представлены в Таблице 3.1.

[3] http://phash.org/
[4] https://instagram.com/natgeo

	aHash	pHash	dHash	gHash
Общее время обработки всех фотографий	2m 40s	19m 6s	2m 36s	2m 38s
Среднее время обработки одной фотографии	0.02s	0.14s	0.019s	0.02s

Таблица 3.1 - Время обработки фотографий

Учитывая медленную скорость работы алгоритма pHash, на следующем этапе будут рассматриваться только алгоритмы gHash, dHash и aHash.

Далее была проведена проверка алгоритмов на коллизии. Опытным путем было выявлено, что расстояние Хэмминга у хэшей похожих изображений меньше 10. Таким образом, если дистанция между хэшами двух уникальных фотографий меньше 10, то алгоритм выдал некорректный результат.

Дистанция	AHashAdaptive	Ghash	Dhash
0	3	0	0
1	3	0	0
2	8	0	2
3	17	1	5
4	40	1	17
5	128	7	61
6	271	25	134
7	459	56	248
8	868	124	462
9	1652	263	859
10	2796	466	1553
11	4789	1019	2705
12	7549	1970	4743
13	11861	3790	8529
14	18537	7146	14889
15	28058	13256	25492

Таблица 3.2 - Количество пар изображений с указанным расстоянием Хэмминга

Ниже приведены графики коллизий для всех рассматриваемых алгоритмов. По оси OX указаны значения расстояний Хэмминга, а по OY - количество пар изображений, соответствующих этим расстояниям.

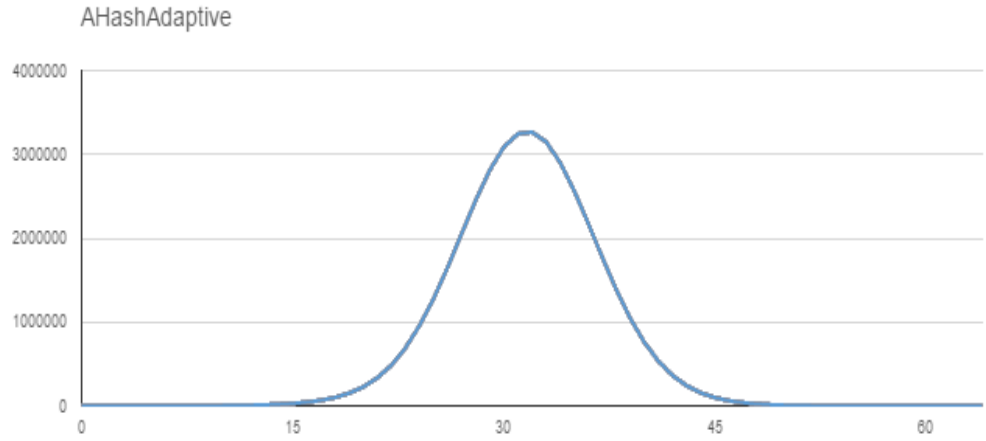

График 3.1 - aHash

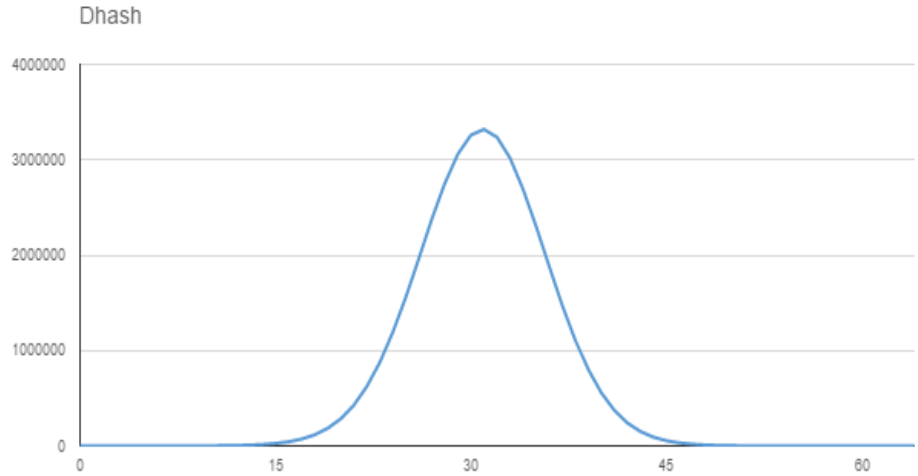

График 3.2 - dHash

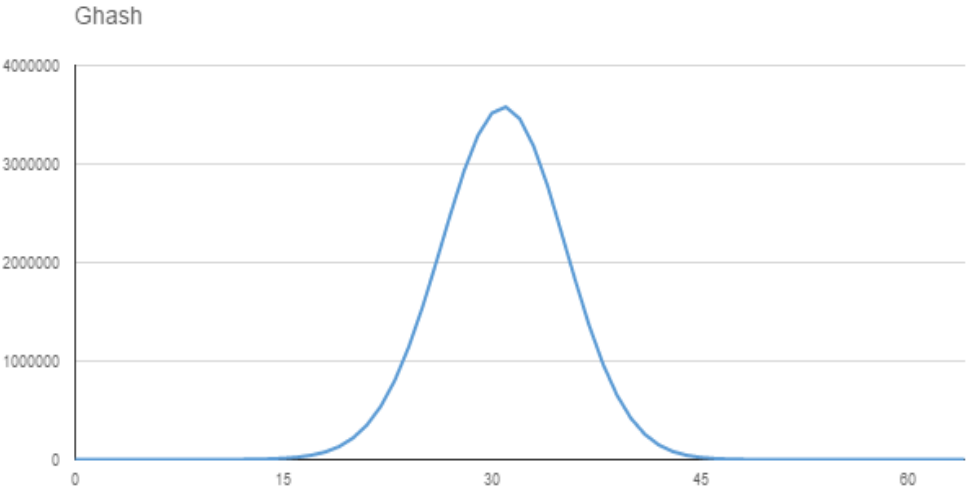

График 3.3 - gHash

Для более наглядной демонстрации разницы между работой алгоритмов был построен общий график, где по оси OX - расстояния

Хэмминга (для наглядности - значения от 0 до 5), а по оси OY - количество пар изображений, между хэшами которых получено данное расстояние.

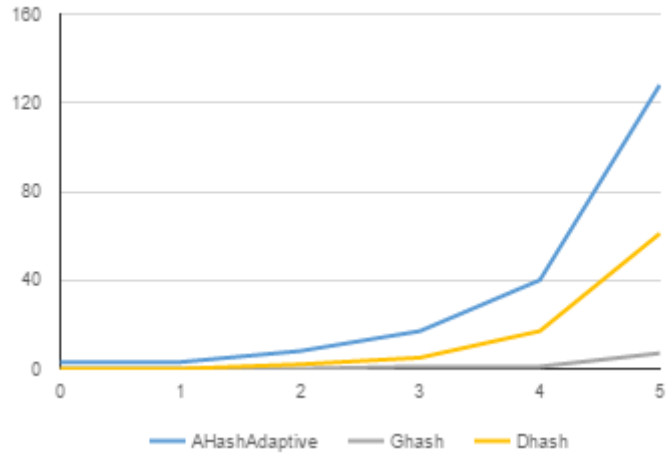

График 3.5 - Сравнение корректности алгоритмов

По результатам работы алгоритмов можно заметить, что gHash более устойчив к коллизиям чем dHash и aHash, что является несомненным преимуществом.

На следующем этапе для 100 изображений из предыдущей коллекции были проведены различные модификации: добавление шумов (noise), размытие (blur), изменения яркости (bright), контрастности (contrast), автоуровень (autolevel), масштабирование (scale), а также была проведена проверка на схожесть оригиналов с их модифицированными версиями.

Дистанция	Noise20	Noise40	blur4px	brightM150	brightP150	contarstP50	contrastM50	autoLevel	Scale	avg
0	48	32	73	16	2	27	19	55	68	37,77777778
1	29	35	23	17	3	35	39	33	26	26,66666667
2	16	17	3	24	9	22	21	7	6	13,88888889
3	5	11	1	17	7	10	5	4	0	6,666666667
4	1	3	0	13	11	5	7	1	0	4,555555556
5	1	0	0	6	11	1	1	0	0	2,222222222
6	0	1	0	2	14	0	4	0	0	2,333333333
7	0	1	0	3	13	0	1	0	0	2
8	0	0	0	2	5	0	3	0	0	1,111111111
9	0	0	0	0	7	0	0	0	0	0,7777777778
10	0	0	0	0	5	0	0	0	0	0,5555555556
11	0	0	0	0	3	0	0	0	0	0,3333333333
12	0	0	0	0	4	0	0	0	0	0,4444444444
13	0	0	0	0	2	0	0	0	0	0,2222222222
14	0	0	0	0	1	0	0	0	0	0,1111111111
15	0	0	0	0	1	0	0	0	0	0,1111111111
16	0	0	0	0	1	0	0	0	0	0,1111111111
17	0	0	0	0	1	0	0	0	0	0,1111111111

Таблица 3.3 - aHash

Дистанция	Noise20	Noise40	blur4px	brightM150	brightP150	contrastP50	contrastM50	autoLevel	Scale	avg
0	33	20	61	10	1	26	6	51	80	32
1	26	28	24	12	7	36	23	31	18	22,77777778
2	18	20	14	26	6	16	34	13	2	16,55555556
3	11	14	1	19	5	12	16	2	0	8,888888889
4	3	6	0	13	12	5	8	2	0	5,444444444
5	2	3	0	10	16	2	5	0	0	4,222222222
6	2	2	0	3	9	3	5	1	0	2,777777778
7	3	3	0	2	14	0	3	0	0	2,777777778
8	1	1	0	4	8	0	0	0	0	1,555555556
9	1	2	0	0	7	0	0	0	0	1,111111111
10	0	1	0	0	4	0	0	0	0	0,5555555556
11	0	0	0	1	4	0	0	0	0	0,5555555556
12	0	0	0	0	3	0	0	0	0	0,3333333333
13	0	0	0	0	2	0	0	0	0	0,2222222222
14	0	0	0	0	0	0	0	0	0	0
15	0	0	0	0	1	0	0	0	0	0,1111111111
16	0	0	0	0	0	0	0	0	0	0
17	0	0	0	0	0	0	0	0	0	0

Таблица 3.4 - dHash

Дистанция	Noise20	Noise40	blur4px	brightM150	brightP150	contrastP50	contrastM50	autoLevel	Scale	avg
0	35	16	39	11	0	28	14	67	81	32,33333333
1	39	28	33	15	4	34	20	27	17	24,11111111
2	13	24	20	23	7	26	28	6	1	16,44444444
3	5	12	5	17	4	8	16	0	1	7,555555556
4	4	12	3	10	14	2	16	0	0	6,777777778
5	0	6	0	12	19	1	4	0	0	4,666666667
6	2	0	0	3	9	1	1	0	0	1,777777778
7	2	1	0	4	8	0	1	0	0	1,777777778
8	0	1	0	1	16	0	0	0	0	2
9	0	0	0	4	4	0	0	0	0	0,888888888
10	0	0	0	0	2	0	0	0	0	0,222222222
11	0	0	0	0	7	0	0	0	0	0,777777777
12	0	0	0	0	2	0	0	0	0	0,222222222
13	0	0	0	0	4	0	0	0	0	0,444444444

Таблица 3.5 - gHash

Согласно результатам тестов, приведенных в таблицах 3.4 – 3.6, gHash показывает точность, сопоставимую с точностью работы алгоритмов aHash и dHash. Учитывая, что при этом gHash показал наибольшую устойчивость к коллизиям, для создания системы проверки фотографий на плагиат был выбран именно он.

Глава 4. Создание веб-сервиса на базе Microsoft Azure

Для реализации веб-сервиса была использована Microsoft Azure - облачная платформа Microsoft, предоставляющая возможность разработки и выполнения приложений и хранения данных на серверах, расположенных в распределённых дата-центрах.

На базах данной платформы и фрэймворка ASP.NET MVC Framework был развернут проект под названием “Instagram Plagiarism Checker”[5] для поиска дубликатов фотографий и была создана база данных для 11000 фотографий из инстаграма natgeo, состоящая из одной таблицы imageHashes вида:

Id	IMGname	IMGhash	User
1	10004091_382952241914058_939407910_n.jpg	-4389679319545026794	natgeo
2	10004098_388871231286627_203400100_n.jpg	5764044603028144142	natgeo
3	10004105_470581853044784_459064896_n.jpg	9056826743112044159	natgeo
4	10004314_665241163543147_614143823_n.jpg	8932750225009034006	natgeo
5	10005144_776225622395677_694181829_n.jpg	7464762228568672199	natgeo
6	10005172_761156830583594_882336968_n.jpg	-2051609205787182721	natgeo
7	10005298_866226493402890_1952871635_n.j...	-4213185629517778717	natgeo
8	10005317_604254926336238_1853447454_n.j...	-3205682362250695485	natgeo
9	10005338_476476549118839_1920068477_n.j...	-54606143376324609	natgeo
10	10005362_1627273517502814_335281251_n.j...	2074464409096543255	natgeo
11	10005419_107305769617038_647852233_n.jpg	-2539776113521272562	natgeo
12	10005437_358847270949638_1934058050_n.j...	-4103499539451673808	natgeo
13	10005491_1485868431625962_1436747319_n...	1552764844228986392	natgeo
14	10005541_336640803176539_1131095407_n.j...	-2755742343996048334	natgeo
15	10005552_790516374394907_419440836_n.jpg	6442256340638285826	natgeo
16	10005576_714265615263053_1549581215_n.j...	-4466481789587458110	natgeo
17	10005663_742811562455971_1293993257_n.j...	-2103246960176083202	natgeo

Рисунок 4.1 - База данных ImageHashes

где

- IMGname - имя изображения
- IMGhash - хэш изображения типа bigint, полученный в результате обработки изображения алгоритмом gHash

[5] http://plagiarismsearch.azurewebsites.net

- User - имя пользователя в Instagram, на страничке которого фотография появилась впервые

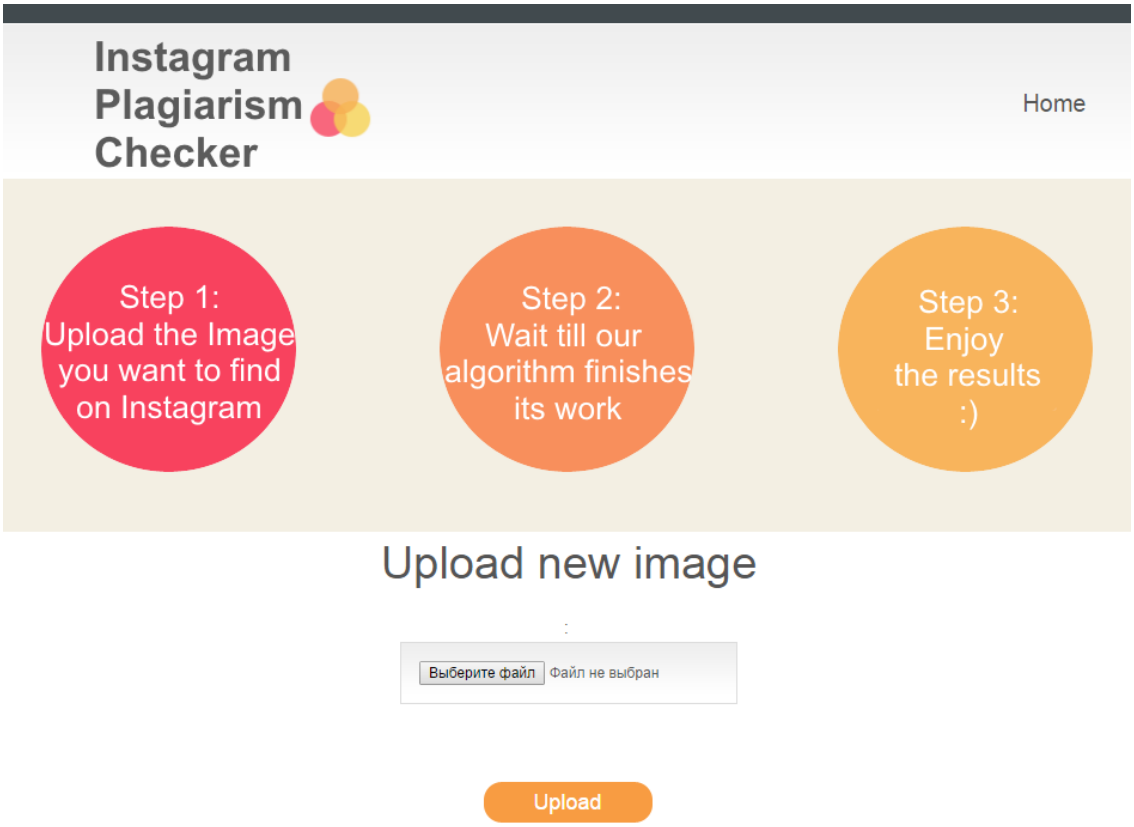

Рисунок 4.2 - Интерфейс веб-сайта

Пользователю предлагается загрузить изображение, дубликат которого он желает найти в инстаграме. После этого алгоритм gHash вычисляет хэш изображения, и затем происходит поиск похожих изображений по базе данных путем вычисления расстояния Хэмминга между хэшами. В итоге пользователю предоставляются все, удовлетворяющие его фотографии, дубликаты.

Необходимо отметить, что фотографии, информация о которых хранится в базе данных, берутся напрямую из инстаграма, так как нецелесообразно хранить бесконечно растущее количество фотографий, например, в облачном хранилище. Таким образом, данный сервис является информационным посредником первого типа. Он лишь предоставляет материал в информационно-телекоммуникационный сети в ответ на запрос пользователя.

С примером работы сервиса можно ознакомиться на рисунках 4.3 и 4.4.

Рисунок 4.3 - Загружаемое изображение

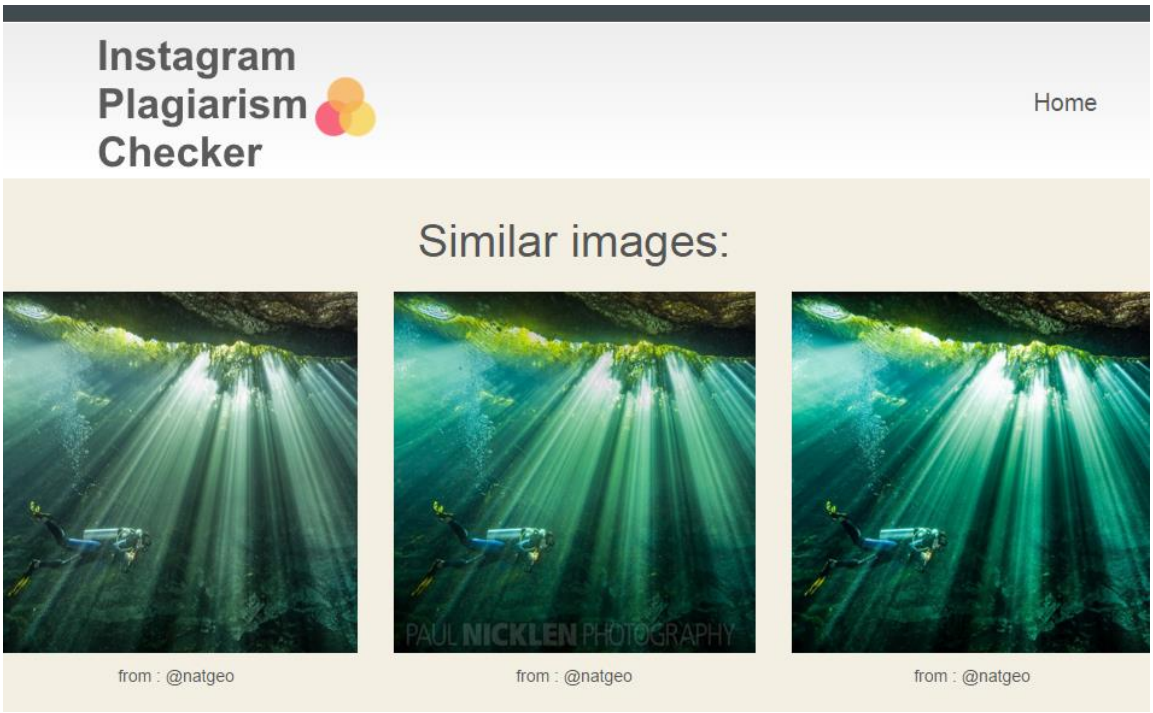

Рисунок 4.4 - Результаты поиска

Выводы

В процессе работы был проведен обзор существующих сервисов поиска похожих изображений и проведены исследования того, какими способами они решают поставленную задачу. Был проведен обзор перцептуальных алгоритмов и создано приложение с удобным графическим интерфейсом для работы с рассмотренными алгоритмами и проведения сравнительных экспериментов, в ходе которых был выбран наиболее эффективный, gHash – авторский алгоритм, показавший себя не хуже других в плане чувствительности к модификациям изображений и являющийся наименее подверженным к коллизиям. Также создан веб-сервис на платформе Microsoft Azure для поиска дубликатов изображений в инстаграме. Для этого была создана база данных для 11000 фотографий из инстаграма natgeo и был предоставлен интуитивно понятный графический интерфейс. В результате были осуществлены основные функции сервиса, такие как: загрузка фотографий, дубликат которой необходимо отыскать, поиск похожих изображений в базе данных, предоставление результатов пользователю с указанием источника.

Заключение

Проблема поиска дубликатов изображений в настоящее время остается одной из самых интересных и по-своему сложных задач. На сегодняшний день не существует такого алгоритма, который бы справился с ней на все 100%. Но современные информационные технологии не стоят на месте и постоянно развиваются. И, кто знает, возможно в недалеком будущем такая проблема будет абсолютно разрешима.

Автор данной ВКР в полной мере справился с поставленной задачей создания системы поиска дубликатов изображений для социальной сети Instagram. В дальнейшем планируется создать систему автозаполнения и обновления базы данных для фотографий из инстаграма и реализовать гибкий поиск в ней. Также возможно применение алгоритмов перцептуального хэширования не только для фотографий, но и для видео и аудио файлов, о чем рассказывается в работах [11] и [12]. Таким образом, станет возможным поиск не только дубликатов фотографий, но и поиск похожих видеозаписей в инстаграме.

Источники и литература

1. Kind of Like That // The Hacker Factor Blog. URL: http://www.hackerfactor.com/blog/?/archives/529-Kind-of-Like-That.html (дата обращения 11.12.2015).
2. Looks Like IT // The Hacker Factor Blog. URL: http://www.hackerfactor.com/blog/index.php?/archives/432-Looks-Like-It.html (дата обращения 11.12.2015).
3. Перцептуальные хэши для сравнения изображений // IT Sector Blog von Alexandr M. URL: http://malexit.ru/?p=93 (дата обращения 13.12.2015).
4. Niu Xia-mu, Jlao Yu-hua An Overview of Perceptual Hashing // ACTA ELECTRONICA SINICA. China, Beijing, July, 2008, Vol.36, No. 7, p. 1405-1411.
5. Zauner C. Implementation and Benchmarking of Perceptual Image Hash Functions. Master's thesis, 2010.
6. Hamming Distance. // Wikipedia. URL: https://en.wikipedia.org/wiki/Hamming_distance (дата обращения 11.12.2015).
7. Сибирь (технология Яндекса) // Wikipedia. URL: https://ru.wikipedia.org/wiki/Сибирь_(технология_Яндекса) (дата обращения 16.04.2016).
8. The Discrete Cosine Transform (DCT) // Cardiff School of Computer Science & Informatics. URL: http://www.cs.cf.ac.uk/Dave/Multimedia/node231.html (дата обращения 11.12.2015).
9. Новый поиск похожих картинок // Блог Яндекса. URL: https://yandex.ru/blog/company/90100 (дата обращения 16.04.2016).
10. R. Venkatesan, S.-M. Koon, M. H. Jakubowski and P. Moulin Robust image hashing // Proc. IEEE ICIP, Vancouver, Canada, September, 2000.

11. M. K. Mıhçak and R. Venkatesan A perceptual audio hashing algorithm: a tool for robust audio identification and information hiding // Inf. Hiding 2001, p. 51-65.
12. O. Kjelsrud Using Perceptual Hash Algorithms to Identify Fragmented and Transformed Video Files, Master's thesis, 2014.
13. Центр документации // Microsoft Azure. URL: https://azure.microsoft.com/ru-ru/documentation/ (дата обращения 22.04.2016).
14. pHash Demo // pHash, the open source perceptual hash library. URL: http://www.phash.org (дата обращения 11.12.2015).

Приложение

Приложение к главам 3 и 4. Реализация алгоритмов aHash, dHash и gHash.

```
1.  using System;
2.  using System.Drawing;
3.  using System.Drawing.Drawing2D;
4.  using System.Drawing.Imaging;
5.  namespace PerceptualHashTester
6.  {
7.
8.      public static class AHashAdaptiveAlg
9.      {
10.
11.         public static Int64 Calculate(Image image)
12.         {
13.             Int64 res = 0;
14.             Bitmap bitmap = null;
15.             bitmap = ImageUtils.ResizeImage(image, 10, 10);
16.             for (int x = 1; x < 9; x++)
17.             {
18.                 for (int y = 1; y < 9; y++)
19.                 {
20.                     byte s1 = ImageUtils.CalculateWeightByAvgOfChanels(bitmap.GetPixel(x
    - 1, y));
21.                     byte s2 = ImageUtils.CalculateWeightByAvgOfChanels(bitmap.GetPixel(x
    - 1, y - 1));
22.                     byte s3 = ImageUtils.CalculateWeightByAvgOfChanels(bitmap.GetPixel(x,
    y - 1));
23.                     byte s4 = ImageUtils.CalculateWeightByAvgOfChanels(bitmap.GetPixel(x
    + 1, y));
24.                     byte s5 = ImageUtils.CalculateWeightByAvgOfChanels(bitmap.GetPixel(x
    + 1, y - 1));
```

```
                    byte sa = (byte)((s1 + s2 + s3 + s4 + s5) / 5f);
                    if (ImageUtils.CalculateWeightByAvgOfChanels(bitmap.GetPixel(x, y)) > sa)
                    {
                        res |= ((Int64)1 << ((y - 1) * 8 + (x - 1)));
                    }
                }
            }
            bitmap.Dispose();
            return res;
        }

    }

public static class GHashAlg
    {
        static float[] row_buffer = new float[32];
        static float[] collumn_buffer = new float[32];

        public static Int64 Calculate(Image image)
        {
            var b32 = ImageUtils.ResizeImage(image, 32, 32);
            for (int x = 0; x < 32; x++)
            {
                float b = 0;
                for (int y = 0; y < 32; y++)
                {
                    b += GetBrith(b32.GetPixel(x, y));
                }
                collumn_buffer[x] = b;
            }
            for (int y = 0; y < 32; y++)
            {
                float b = 0;
```

```
58.                 for (int x = 0; x < 32; x++)
59.                 {
60.                     b += GetBrith(b32.GetPixel(x, y));
61.                 }
62.                 row_buffer[y] = b;
63.             }
64.             Int64 res = 0;
65.             for (int i = 0; i < 32; i++)
66.             {
67.                 if (row_buffer[i] > row_buffer[(i + 1) % 32]) res |= (Int64)1 << i;
68.                 if (collumn_buffer[i] > collumn_buffer[(i + 1) % 32]) res |= (Int64)1 <<
    (i + 32);
69.             }
70.             b32.Dispose();
71.             return res;
72.         }
73.         static float GetBrith(Color color)
74.         {
75.             return (color.R + color.G + color.B) / 3f;
76.         }
77.     }
78.
79.     public static class DHashAlg
80.     {
81.         public static Int64 Calculate(Image image)
82.         {
83.             Int64 res = 0;
84.             var bitmap = ImageUtils.ResizeImage(image, 9, 8);
85.             int i = 0;
86.             for (int y = 0; y < 8; y++)
87.             {
88.                 for (int x = 0; x < 8; x++)
89.                 {
90.                     Color c1 = bitmap.GetPixel(x, y);
```

```
                    byte a1 = ImageUtils.CalculateWeightByAvgOfChanels(c1);
                    Color c2 = bitmap.GetPixel(x + 1, y);
                    byte a2 = ImageUtils.CalculateWeightByAvgOfChanels(c2);
                    if (a1 > a2)
                    {
                        res |= ((Int64)1 << i);
                    }
                    i++;
                }
            }
            return res;
        }

    }
    public static class FastFeatures
    {
        public struct FPoint
        {
            public byte _x;
            public byte _y;
            public int _w;
        }
        private static int[] offsetX = new int[] { 0, 1, 2, 3, 3, 3, 2, 1, 0, -1, -2, -
3, -3, -3, -2, -1 };
        private static int[] offsetY = new int[] { 3, 3, 2, 1, 0, -1, -2, -3, -3, -3, -
2, -1, 0, 1, 2, 3 };
        public static Bitmap Do(Bitmap bitmap, FPoint[] res)
        {

            Bitmap rBitmap = null;
            if (bitmap.Width > bitmap.Height)
            {
                rBitmap = ImageUtils.ResizeImage(bitmap, 256,
(int)((float)bitmap.Height / (float)bitmap.Width * 256));
```

```
122.             }
123.             else
124.             {
125.                 rBitmap = ImageUtils.ResizeImage(bitmap, 256, (int)((float)bitmap.Width
    / (float)bitmap.Height * 256f));
126.             }
127.             for (int x = 3; x < rBitmap.Width - 3; x++)
128.             {
129.                 for (int y = 3; y < rBitmap.Height - 3; y++)
130.                 {
131.                     int w = GetPixelWeight(rBitmap, x, y);
132.                     int maxI = -1;
133.                     for (int i = 0; i < res.Length; i++)
134.                     {
135.                         if (w > res[i]._w)
136.                         {
137.                             maxI = i;
138.                         }
139.                     }
140.                     if (maxI != -1)
141.                     {
142.                         for (int i = 1; i <= maxI; i++)
143.                         {
144.                             res[i - 1] = res[i];
145.                         }
146.                         res[maxI] = new FPoint() { _w = w, _x = (byte)x, _y = (byte)y
    };
147.                     }
148.                  }
149.             }
150.             for (int i = 0; i < res.Length; i++)
151.             {
152.                 rBitmap.SetPixel(res[i]._x, res[i]._y, Color.Red);
153.                 for (int j = 0; j < 16; j++)
154.                 {
```

```
155.                     int oX = offsetX[j];
156.                     int oY = offsetY[j];
157.                     rBitmap.SetPixel(res[i]._x + oX, res[i]._y + oY, Color.Green);
158.                 }
159.             }
160.             return rBitmap;
161.         }
162.         private static int GetPixelWeight(Bitmap bitmap, int x, int y)
163.         {
164.             int b = 0;
165.             Color currentColor = bitmap.GetPixel(x, y);
166.             int currentB = (currentColor.R + currentColor.B + currentColor.G) / 3;
167.             for (int i = 0; i < 16; i++)
168.             {
169.                 int oX = offsetX[i];
170.                 int oY = offsetY[i];
171.                 Color c = bitmap.GetPixel(x + oX, y + oY);
172.                 var nb = ((c.R + c.G + c.B) / 3 - currentB);
173.                 b += nb;
174.             }
175.             return b;
176.
177.         }
178.     }
179.     public static class ImageUtils
180.     {
181.         public static Bitmap ResizeImage(Image image, int width, int height)
182.         {
183.             var destRect = new Rectangle(0, 0, width, height);
184.             var destImage = new Bitmap(width, height);
185.
186.             destImage.SetResolution(image.HorizontalResolution,
    image.VerticalResolution);
187.
```

```
188.             using (var graphics = Graphics.FromImage(destImage))
189.             {
190.                 graphics.CompositingMode = CompositingMode.SourceCopy;
191.                 graphics.CompositingQuality = CompositingQuality.HighQuality;
192.                 graphics.InterpolationMode = InterpolationMode.HighQualityBicubic;
193.                 graphics.SmoothingMode = SmoothingMode.AntiAlias;
194.                 graphics.PixelOffsetMode = PixelOffsetMode.HighQuality;
195. 
196.                 using (var wrapMode = new ImageAttributes())
197.                 {
198.                     wrapMode.SetWrapMode(WrapMode.TileFlipXY);
199.                     graphics.DrawImage(image, destRect, 0, 0, image.Width,
    image.Height, GraphicsUnit.Pixel, new ImageAttributes());
200.                 }
201.             }
202. 
203.             return destImage;
204.         }
205.         public static byte CalculateWeightByAvgOfChanels(Color c)
206.         {
207.             return (byte)((c.R + c.G + c.B) / 3f);
208.         }
209.     }
210. }
```